一、 计算机网络

1.1网络分层

1. 物理层

建立物理连接，传输比特流数据，光纤、电缆属于物理层

2. 数据链路层

在物理层的基础上建立逻辑连接，通过物理地址寻址，进行差错检验，传输数据帧交换机属于数据链路层

3. 网络层

建立网络连接，通过IP地址寻址，路由器属于网络层

4. 传输层

介于网络层和会话层之间，从会话层接收数据，将数据分成数据段segment，TCP、UDP属于传输层

5. 会话层

建立会话和维持会话。会话管理

6. 表示层

数据压缩、数据解压、数据加密解密、数据编码转换

7. 应用层

操作系统或网络应用程序与网络提供网络访问接口，Telnet,FTP,SMTP,HTTP,DNS

1.2 底层网络协议

1. ARP 地址解析协议

通过IP地址查询MAC地址

2. ICMP 互联网控制消息协议

它是TCP/IP协议族的一个子协议，属于网络层协议，面向无连接的，主要用于在主机与路由器之间传递控制信息，包括报告错误、交换受限控制和状态信息等。当遇到IP数据无法访问目标、IP路由器无法按当前的传输速率转发数据包等情况时，会自动发送ICMP消息。

3. DHCP 动态主机配置协议

用于内部网络或网络服务供应商自动分配IP地址给用户

用于内部网络管理员作为对所有电脑作集中管理的手段

1.2 TCP 传输控制协议

TCP是面向连接的传输层协议。

TCP连接是点对点的（套接字--IP:Port到套接字）。

TCP提供可靠交付

的服务。

TCP提供全双工通信。

面向字节流。

面向连接的、可靠的、基于字节流的传输层协议

TCP三次握手的过程如下：

1. 客户端发送SYN（SEQ=x）报文给服务器端，进入SYN\_SEND状态。

2. 服务器端收到SYN报文，回应一个SYN （SEQ=y）ACK(ACK=x+1）报文，进入SYN\_RECV状态。

3. 客户端收到服务器端的SYN报文，回应一个ACK(ACK=y+1）报文，进入Established状态。

三次握手完成，TCP客户端和服务器端成功地建立连接，可以开始传输数据了。

四次挥手关闭连接：

之所以需要四次是因为TCP是全双工的，因此每个方向必须单独关闭。

FIN:结束标志

ACK：确认标志

1.3 UDP 用户数据报协议

面向无连接的，不可靠的传输层协议

延迟小，数据传输效率高

1.4 IP

A类地址：1.0.0.0---126.0.0.0 10.X.X.X是私有地址

B类地址：128.0.0.0---191.255.0.0 172.16.0.0---172.31.255.255是私有地址

C类地址：192.0.0.0---223.255.255.0 192.168.X.X是私有地址

D类地址：224.0.0.0---239.255.255.255

E类地址：240.0.0.0---255.255.255.254

127.X.X.X是保留地址，用做循环测试用的

1.5 HTTP

HTTP构建于TCP/IP协议之上，默认端口号是80。

HTTP是无连接无状态的。

无连接: 限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。后来使用了Keep-Alive技术。

无状态: 协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。即我们给服务器发送 HTTP 请求之后，服务器根据请求，会给我们发送数据过来，但是，发送完，不会记录任何信息。

HTTP 协议这种特性有优点也有缺点，优点在于解放了服务器，每一次请求“点到为止”不会造成不必要连接占用，缺点在于每次请求会传输大量重复的内容信息。

为了解决HTTP无状态的缺点，两种用于保持 HTTP 连接状态的技术就应运而生了，一个是 Cookie，而另一个则是 Session。Cookie在客户端记录状态，比如登录状态。Session在服务器记录状态。

HTTP 请求报文头部

User-Agent：产生请求的浏览器类型。

Accept：客户端可识别的响应内容类型列表;

Accept-Language：客户端可接受的自然语言;

Accept-Encoding：客户端可接受的编码压缩格式;

Accept-Charset：可接受的应答的字符集;

Host：请求的主机名，允许多个域名同处一个IP 地址，即虚拟主机;

Connection：连接方式(close 或 keep-alive);

Cookie：存储于客户端扩展字段，向同一域名的服务端发送属于该域的cookie;

请求包体：在POST方法中使用。

Referer：包含一个URL，用户从该URL代表的页面出发访问当前请求的页面。

If-Modified-Since：文档的最后改动时间

HTTP 响应头

Allow 服务器支持哪些请求方法（如GET、POST等）。

Content-Encoding 文档的编码（Encode）方法。

Content-Length 表示内容长度。只有当浏览器使用持久HTTP连接时才需要这个数据。

Content-Type 表示后面的文档属于什么MIME类型。

Date 当前的GMT时间。你可以用setDateHeader来设置这个头以避免转换时间格式的麻烦。

Expires 应该在什么时候认为文档已经过期，从而不再缓存它。

Last-Modified 文档的最后改动时间。

Refresh 表示浏览器应该在多少时间之后刷新文档，以秒计。

Server 服务器名字。

Set-Cookie 设置和页面关联的Cookie。

ETag：被请求变量的实体值。ETag是一个可以与Web资源关联的记号（MD5值）。

Cache-Control：这个字段用于指定所有缓存机制在整个请求/响应链中必须服从的指令。

Http request的几种类型。

GET 请求指定的页面信息，并返回实体主体。

POST 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。

PUT 从客户端向服务器传送的数据取代指定的文档的内容。

DELETE 请求服务器删除指定的页面。

GET 可提交的数据量受到URL长度的限制，HTTP协议规范没有对URL长度进行限制。这个限制是特定的浏览器及服务器对它的限制

理论上讲，POST是没有大小限制的，HTTP协议规范也没有进行大小限制，出于安全考虑，服务器软件在实现时会做一定限制.

HTTP Keep-Alive 简单说就是保持当前的TCP连接，避免了重新建立连接。

HTTP 长连接不可能一直保持，例如 Keep-Alive: timeout=5, max=100，表示这个TCP通道可以保持5秒，max=100，表示这个长连接最多接收100次请求就断开。

1.6 面试题：

1. CSMA/CD有什么作用？

CSMA/CD即带冲突检测的载波监听多路访问技术，应用在 OSI 的第二层数据链路层，是为了解决共享介质的传输效率的问题。其原理简单总结为：先听后发，边发边听，冲突停发，随机延迟后重发。

2. Http会话的过程？

建立tcp连接

发出请求文档

发出响应文档

释放tcp连接

3.TCP协议如何实现可靠传输？

TCP 协议是通过ARQ协议以及等待、确认、重传等机制实现可靠传输。

二、 数据结构与算法

1.1 二叉树

1. 先序遍历。DLR

2. 中序遍历。LDR

3. 后序遍历。LRD

始终都是先从左至右，先序，中序，后序指的是根结点。

已知先序和中序，或者后序和中序遍历可以确定树的结构，没有中序遍历无法确定树的结构，因为无法确定左子树和右子树。

已知先序和中序确定树结构：

1. 先序的第一个结点是根结点

2. 从中序中找到该结点，结点左边为左子树，右边为右字树

3. 依次1,2步骤

1.2 霍夫曼树

1. 霍夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度（若根结点为0层，叶结点到根结点的路径长度为叶结点的层数）。树的路径长度是从树根到每一结点的路径长度之和，记为WPL=（W1L1+W2L2+W3L3+...+WnLn），N个权值Wi（i=1,2,...n）构成一棵有N个叶结点的二叉树，相应的叶结点的路径长度为Li（i=1,2,...n）。

2. 霍夫曼树构造：

假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，则哈夫曼树的构造规则为：

(1) 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3)从森林中删除选取的两棵树，并将新树加入森林；

(4)重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

3. 霍夫曼编码

对于一个霍夫曼树，所有左链接取'0'、右链接取'1'。从树根至树叶依序记录所有字母的编码。

4. 带权路径

结点的权：若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。

结点的带权路径：从根结点到该结点之间的路径长度与该结点的权的乘积。

树的带权路径：所有叶子结点的带权路径长度之和，记为WPL

5. 二叉搜索树

任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

任意节点的左、右子树也分别为二叉查找树；

没有键值相等的节点。

二分查找的时间复杂度是O(log(n))，最坏情况下的时间复杂度是O(n)（相当于顺序查找）

6. 平衡二叉树

平衡树是计算机科学中的一类改进的二叉查找树。一般的二叉查找树的查询复杂度是跟目标结点到树根的距离（即深度）有关，因此当结点的深度普遍较大时，查询的均摊复杂度会上升，为了更高效的查询，平衡树应运而生了。平衡指所有叶子的深度趋于平衡，更广义的是指在树上所有可能查找的均摊复杂度偏低。

AVL树是最先发明的 自平衡二叉查找树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。

它的左子树和右子树都是平衡二叉树。

左子树和右子树的深度之差的绝对值不超过1。

增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

右旋：左结点转到根节点位置。

左旋：右节点转到根节点位置。

高度为k的AVL树，节点数N最多2^k -1，即满二叉树；

7. 红黑树

节点是红色或黑色。

根是黑色。

所有叶子都是黑色（叶子是NIL节点）。

每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）

从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

红黑树的插入和删除操作会导致不再符合红黑树的性质。恢复红黑树的性质需要少量（O(log n)）的颜色变更（实际是非常快速的）和不超过三次树旋转（对于插入操作是两次）。虽然插入和删除很复杂，但操作时间仍可以保持为O(log n)次。

8. B树

是一种多路搜索树。

1）所有叶子结点位于同一层，并且 不带信息。

2）树中每个节点最多有m个子树(即至多含有m-1个关键字)。

3）若根节点不是终端节点，则根节点子树[2,m].

4）除根节点外其他非叶子节点至少有[m/2]个子树(即至少含有[m/2]-1个关键字)。

每个非叶子节点的结构为：

| n | p0 | k1 | p1 | k2 | p2 | ... | kn | pn |

n为该节点中的关键字个数，除根节点外，其他所有非叶子节点的关键字个数n：[m/2]-1 <= n <= m-1;

ki(i <= i <=n)为该节点的关键字且满足ki < ki+1

pi(0 <= i <=n)为该节点的孩子节点指针pi(0 <= i <=n-1)所指节点上的关键字大于等于ki且小于ki+1，pn所指节点上的关键字大于kn.

B-树的阶：所有节点的孩子节点数的最大值。

B树的查找：

1）若k=key[i]，则查找成功；

2）若k<key[1]，则沿指针ptr[0]所指的子树继续查找；

3）若key[i]<k<key[i+1]，则沿着指针ptr[i]所指的子树继续查找；

4）若k>key[n]，则沿着指针ptr[n]所指的子树继续查找。

B树的插入：

插入后如果不满足条件要分裂，将中间的结点升到父节点，然后平分其余结点，直至满足条件为止。

B树的删除：

首先查找B树中需删除的元素，如果该元素在B树中存在，则将该元素在其结点中进行删除；如果删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素到父节点中，然后是移动之后的情况；如果没有，直接删除后，然后是移动之后的情况。

删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于Min(m/2)-1，则需要看其某相邻兄弟结点是否丰满，如果丰满，则向父节点借一个元素来满足条件；如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于Min(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点。

9. B+树

是一种自平衡二叉树，通常用于数据库和操作系统的文件系统中。B+树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+树元素自底向上插入，这与二叉树恰好相反。B+树不需要象其他自平衡二叉查找树那样经常的重新平衡。

与B树的不同点：

1）有n个子树的节点有n个关键字

2）所有叶子节点包含全部关键字及指向相应记录的指针，而且叶子节点按关键字大小顺序链接(可以把每个叶子及诶单看成一个基本索引块，它的指针不再指向另一级索引块，而是直接指向数据文件中的记录)

所有分支节点中仅仅包含它的哥哥子节点(即下级索引块)中最大关键字及指向子节点的指针。

3）B+树种所有叶子节点包含了全部关键字，即其他非叶子节点中的关键字包含在叶子节点中，而在B-树中，关键字是不重复的。

为什么说B+树比B-树更适合实际应用中操作系统的文件索引和数据库索引？

1. B+树的磁盘读写代价更低

我们都知道磁盘时可以块存储的，也就是同一个磁道上同一盘块中的所有数据都可以一次全部读取。而B+树的内部结点并没有指向关键字具体信息的指针(比如文件内容的具体地址 ） 。因此其内部结点相对B-树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。这样，一次性读入内存中的需要查找的关键字也就越多。\*\*相对来说IO读写次数也就降低了\*\*。

举个例子，假设磁盘中的一个盘块容纳`16bytes`，而一个关键字`2bytes`，一个关键字具体信息指针`2bytes`。一棵9阶B-树(一个结点最多8个关键字)的内部结点需要2个盘块。而B+树内部结点只需要1个盘块。\*\*当需要把内部结点读入内存中的时候，B-树就比B+数多一次盘块查找时间（在磁盘中就是盘片旋转的时间）\*\*。

2. B+树的查询效率更加稳定。

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。\*\*所有关键字查询的路径长度相同，导致每一个数据的查询效率相当\*\*。

10. B\*树

关键字的数量要多一些，分裂次数更少，节点空间使用率更高

每一次演变都是为了让节点的空间更合理的运用，从而使树的层级减少到达快速查找的目的。

三、 Java 基础

1. 封装

将对象属性和操作合成一个整体，尽可能的隐藏内部实现，对外开放访问的接口。减少外部对内部的操作，提高代码的安全性。

2. 多态

基于对象所属类的不同，外部对同一个方法的调用，实际执行的逻辑不同。很显然，多态实际上是依附于继承的第二种含义的。

A. 要有继承

B. 要有方法重写

C. 父类引用指向子类

3. 继承

1） 集成基类，拥有基类的属性，都是进行扩展。

2） 声明某个子类兼容于某基类（或者说，接口上完全兼容于基类），外部调用者可无需关注其差别（内部机制会自动把请求派发dispatch到合适的逻辑）。

4. 重写

子类重写父类方法，只有实例方法可以被重写，重写后的方法必须仍为实例方法。成员变量和静态方法都不能被重写，只能被隐藏。

方法的重写（override）两同两小一大原则：

1. 方法名相同，参数类型相同

2. 子类返回类型小于等于父类方法返回类型

3. 子类抛出异常小于等于父类方法抛出异常

4. 子类访问权限大于等于父类方法访问权限

5. 重载

在同一个类中，有多个方法名相同，参数列表不同（参数个数不同，参数类型不同），与方法的返回值无关，与权限修饰符无关。编译器通过对方法签名的识别即可静态编译出不同的方法。这也是java中重载与重写的区别之一。

Java中方法是可以和类名同名的，和构造方法唯一的区别就是，构造方法没有返回值。

6. 集合框架

Java集合框架提供了数据持有对象的方式，提供了对数据集合的操作。Java集合框架位于java.util包下，主要有三个大类：Collection、Map接口以及对集合进行操作的工具类。

1） Collection

ArrayList：线程不同步。默认初始容量为10，当数组大小不足时增长率为当前长度的50%。内部采用数组实现。实现List接口。

Vector：线程同步。默认初始容量为10，当数组大小不足时增长率为当前长度的100%。它的同步是通过加synchronized实现的。Add方法有加synchronized.

LinkedList：线程不同步。双端队列形式。实现List接口和Deque。内部采用链表实现。

Stack：线程同步。继承自Vector，添加了几个方法来完成栈的功能。

Set：Set是一种不包含重复元素的Collection，Set最多只有一个null元素。

HashSet：线程不同步，内部使用HashMap进行数据存储，提供的方法基本都是调用HashMap的方法，所以两者本质是一样的。集合元素可以为NULL。Set保存的值作为Map中的键进行保存。

public HashSet() {

map = new HashMap<>();

}

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

LinkedHashSet 继承自HashSet，内部采用LinkedHashMap实现

NavigableSet：添加了搜索功能，可以对给定元素进行搜索：小于、小于等于、大于、大于等于，放回一个符合条件的最接近给定元素的 key。

TreeSet：线程不同步，内部使用NavigableMap操作。默认元素“自然顺序”排列，可以通过Comparator改变排序。采用TreeMap实现。

EnumSet：线程不同步。内部使用Enum数组实现，速度比HashSet快。只能存储在构造函数传入的枚举类的枚举值。

2） Map

HashMap：线程不同步。根据key的hashcode进行存储，内部使用静态内部类Node的数组进行存储，默认初始大小为16，每次扩大一倍。当发生Hash冲突时，采用拉链法（链表）。可以接受为null的键值(key)和值(value)。Null键的hashCode为0。JDK 1.8中：当单个桶中元素个数大于等于8时，链表实现改为红黑树实现；当元素个数小于6时，变回链表实现。由此来防止hashCode攻击。

HashMap 结构：

多个Entry数组，每一个Entry又是一个链表。

创建HashMap时，默认初始大小为16，如果创建时指定大小，则创建的大小为大于指定大小的最小的2次方幂。比如指定大小为为7，则创建的实际大小为8。保证是2次方幂的计算过程：

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

2次方幂的原因是：便于计算数组的索引。

在put数据时，先计算出Key的hash值，再根据hash值计算出数组的索引，传统的方法是用hash mod n，n为数组的长度。但是这样子效率不高，jdk1.6后改为

(n - 1) & hash。

LinkedHashMap：保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的. 也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。

TreeMap：线程不同步，基于 红黑树 （Red-Black tree）的NavigableMap 实现，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。

HashTable：线程安全，继承自Dictionary，默认初始大小为11，之后每次扩充2n+1, HashMap的迭代器(Iterator)是fail-fast迭代器。HashTable不能存储NULL的key和value。已经被废弃。如果不需要线程同步，使用HashMap，如果需要使用ConcurrentHashMap.

ConcurrentHashMap

ConcurrentHashMap 实现了ConcurrentMap接口，ConcurrentMap继承Map。

ConcurrentHashMap的锁分段技术：假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术。首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap不允许Key或者Value的值为NULL。

Collections、Arrays：集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

Comparable、Comparator：一般是用于对象的比较来实现排序，两者略有区别。

Comparable是在集合内部定义的方法实现的排序，Comparator是在集合外部实现的排序。所以如果想实现排序，就需要在集合外定义Comparator接口的方法或在集合内实现Comparable接口的方法。

Comparable是一个对象本身就已经支持自比较所需要实现的接口，如String、Integer自己就实现了Comparable接口，可完成比较大小操作。自定义类要在加入list容器中后能够排序，也可以实现Comparable接口，在用Collections类的sort方法排序时若不指定Comparator，那就以自然顺序排序。所谓自然顺序就是实现Comparable接口设定的排序方式。

Comparator是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足要求时，可写一个比较器来完成两个对象之间大小的比较。Comparator体现了一种策略模式(strategy design pattern)，就是不改变对象自身，而用一个策略对象(strategy object)来改变它的行为。

类设计者没有考虑到比较问题而没有实现Comparable接口。这是我们就可以通过使用Comparator，这种情况下，我们是不需要改变对象的。

一个集合中，我们可能需要有多重的排序标准，这时候如果使用Comparable就有些捉襟见肘了，可以自己继承Comparator提供多种标准的比较器进行排序。

JDK 1.8中

将原先 table数组＋单向链表 的数据结构，变更为 table数组＋单向链表＋红黑树 的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。

Fail-fast 迭代器：

fail-fast机制在遍历一个集合时，当集合结构被修改，会抛出Concurrent Modification Exception。

迭代器在遍历过程中是直接访问内部数据的，因此内部的数据在遍历的过程中无法被修改。为了保证不被修改，迭代器内部维护了一个标记 “mode” ，当集合结构改变（添加删除或者修改），标记"mode"会被修改，而迭代器每次的hasNext()和next()方法都会检查该"mode"是否被改变，当检测到被修改时，抛出Concurrent Modification Exception。

final void checkForComodification() {

if (ArrayList.this.modCount == this.expectedModCount)

return;

throw new ConcurrentModificationException(); }

Fail-safe 迭代器：ConcurrentHashMap

fail-safe任何对集合结构的修改都会在一个复制的集合上进行修改，因此不会抛出ConcurrentModificationException。

fail-safe机制有两个问题

（1）需要复制集合，产生大量的无效对象，开销大

（2）无法保证读取的数据是目前原始数据结构中的数据。

7. Hash

哈希表（Hash Table，也叫散列表），是根据关键码值 (Key-Value) 而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。哈希表的实现主要需要解决两个问题，哈希函数和冲突解决。

1) 哈希函数 生成HashCode

哈希函数也叫散列函数，它对不同的输出值得到一个固定长度的消息摘要。理想的哈希函数对于不同的输入应该产生不同的结构，同时散列结果应当具有同一性（输出值尽量均匀）和雪崩效应（微小的输入值变化使得输出值发生巨大的变化）。

2) 冲突解决

开放地址法：以发生冲突的哈希地址为输入，通过某种哈希冲突函数得到一个新的空闲的哈希地址的方法。有以下几种方式：

线性探查法：从发生冲突的地址开始，依次探查下一个地址，直到找到一个空闲单元。

平方探查法：设冲突地址为d0，则探查序列为：d0+1^2,d0-1^2,d0+2^2...

拉链法：把所有的同义词用单链表链接起来。在这种方法下，哈希表每个单元中存放的不再是元素本身，而是相应同义词单链表的头指针。HashMap就是使用这种方法解决冲突的。

8. Java 分派机制

我们都是知道java文件都需要编译成class文件，而一切方法调用在class文件里存储的都是符号引用，而不是方法的实际运行时内存布局的入口地址（相当于直接引用）。在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析成立的前提是：方法的程序真正运行之前就有一个可确认的调用版本，并且这个方法的调用版本在运行期是不可变的。换句话说，调用目标在程序代码写好、编辑器进行编译时就必须确认下来，这类方法调用的调用称为解析。

在Java虚拟机里提供了5条调用方法字节码指令，分别如下。

invokestatic：调用静态方法

invokespeciak： 调用实例构造器<init>方法、私用方法和父类方法

invokevirtual： 调用所有的虚方法

invokeinterface：调用接口时，会在运行再确定一个实现接口的对象

invokedynamic：现在运行时动态解析出调用点限定符引用的方法，再执行方法

只有被invokestatic和invokespecial指令调用的方法，可以在解析阶段中确定调用的版本，符合这个条件的静态方法、私有方法、实例构造器、父类方法。它们在类加载的解析时候就会把符号引用解析为直接引用。这些方法被称为非虚方法。解析调用一定是一个静态的过程，在编译期间就完全确定，而分配调用可能是静态的也可能是动态的。

1） 静态分配

People man = new Man()

静态分派主要针对重载，方法调用时如何选择。在上面的代码中，People被称为变量的引用类型，Man被称为变量的实际类型。静态类型是在编译时可知的，而动态类型是在运行时可知的，编译器不能知道一个变量的实际类型是什么。

编译器在重载时候通过参数的静态类型而不是实际类型作为判断依据。并且静态类型在编译时是可知的，所以编译器根据重载的参数的静态类型进行方法选择。

在某些情况下有多个重载，那编译器如何选择呢？ 编译器会选择"最合适"的函数版本，那么怎么判断"最合适“呢？越接近传入参数的类型，越容易被调用。

所有依赖静态类型来定位方法执行版本的分派动作称为静态分配，静态分配的典型动作是方法重载，静态分派发生在编译阶段，虽然编译器能确定方法的重载版本，但是很多情况下这个重载的版本并不是“唯一的”，往往只能确定一个“更加合适的”版本。

2） 动态分配

动态分派主要针对重写，使用invokevirtual指令调用。invokevirtual指令多态查找过程：

A. 找到操作数栈顶的第一个元素所指向的对象的实际类型，记为C。

B. 如果在类型C中找到与常量中的描述符合简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果权限校验不通过，返回java.lang.IllegalAccessError异常。

C. 否则，按照继承关系从下往上一次对C的各个父类进行第2步的搜索和验证过程。

D. 如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError异常。

3） 虚拟机动态分派的实现

由于动态分派是非常繁琐的动作，而且动态分派的方法版本选择需要考虑运行时在类的方法元数据中搜索合适的目标方法，因此在虚拟机的实现中基于性能的考虑，在方法区中建立一个虚方法表（invokeinterface有接口方法表），来提高性能。

虚方法表中存放各个方法的实际入口地址。如果某个方法在子类没有重写，那么子类的虚方法表里的入口和父类入口一致，如果子类重写了这个方法，那么子类方法表中的地址会被替换为子类实现版本的入口地址。

9. Java异常

Java标准库内建了一些通用的异常，这些类以Throwable为顶层父类。

Throwable又派生出Error类和Exception类。

错误：Error类以及他的子类的实例，代表了JVM本身的错误。错误不能被程序员通过代码处理，Error很少出现。因此，程序员应该关注Exception为父类的分支下的各种异常类。

异常：Exception以及他的子类，代表程序运行时发送的各种不期望发生的事件。可以被Java异常处理机制使用，是异常处理的核心。

非检查异常（unckecked exception）：Error 和 RuntimeException 以及他们的子类。javac在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常。所以如果愿意，我们可以编写代码处理（使用try…catch…finally）这样的异常，也可以不处理。对于这些异常，我们应该修正代码，而不是去通过异常处理器处理 。这样的异常发生的原因多半是代码写的有问题。如除0错误ArithmeticException，错误的强制类型转换错误ClassCastException，数组索引越界ArrayIndexOutOfBoundsException，使用了空对象NullPointerException等等。

检查异常（checked exception）：除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作（使用try…catch…finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException , IOException,ClassNotFoundException 等。

处理异常：

在编写代码处理异常时，对于检查异常，有2种不同的处理方式：使用try…catch…finally语句块处理它。或者，在函数签名中使用throws 声明交给函数调用者caller去解决。

//finally块通常是可选的。

//无论异常是否发生，异常是否匹配被处理，finally都会执行。

在 try块中即便有return，break，continue等改变执行流的语句，finally也会执行。

//一个try至少要有一个catch块，否则，至少要有1个finally块。但是finally不是用来处理异常的，finally不会捕获异常。

//finally主要做一些清理工作，如流的关闭，数据库连接的关闭等。

try块中的局部变量和catch块中的局部变量（包括异常变量），以及finally中的局部变量，他们之间不可共享使用。

每一个catch块用于处理一个异常。异常匹配是按照catch块的顺序从上往下寻找的，只有第一个匹配的catch会得到执行。匹配时，不仅运行精确匹配，也支持父类匹配，因此，如果同一个try块下的多个catch异常类型有父子关系，应该将子类异常放在前面，父类异常放在后面，这样保证每个catch块都有存在的意义.

自定义异常：

如果要自定义异常类，则扩展Exception类即可，因此这样的自定义异常都属于检查异常（checked exception）。如果要自定义非检查异常，则扩展自RuntimeException。

10. Java泛型

1） 类型擦除

Java中的泛型基本上都是在编译器这个层次来实现的，在生成的Java字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉，这个过程就称为类型擦除。如在代码中定义的List<Object>和List<String>等类型，在编译之后都会变成List。JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的。

比如一个方法如果接收List<Object>作为形式参数，那么如果尝试将一个List<String>的对象作为实际参数传进去，却发现无法通过编译。虽然从直觉上来说，Object是String的父类，这种类型转换应该是合理的。但是实际上这会产生隐含的类型转换问题，因此编译器直接就禁止这样的行为。

A. 泛型类并没有自己独有的Class类对象。比如并不存在List<String>.class或是List<Integer>.class，而只有List.class。

B. 静态变量是被泛型类的所有实例所共享的。对于声明为MyClass<T>的类，访问其中的静态变量的方法仍然是 MyClass.myStaticVar。不管是通过new MyClass<String>还是new MyClass<Integer>创建的对象，都是共享一个静态变量。

C. 泛型的类型参数不能用在Java异常处理的catch语句中。因为异常处理是由JVM在运行时刻来进行的。由于类型信息被擦除，JVM是无法区分两个异常类型MyException<String>和MyException<Integer>的。对于JVM来说，它们都是 MyException类型的。也就无法执行与异常对应的catch语句。

2） 通配符

在使用泛型类的时候，既可以指定一个具体的类型，如List<String>就声明了具体的类型是String；也可以用通配符?来表示未知类型，如List<?>就声明了List中包含的元素类型是未知的。 通配符所代表的其实是一组类型，但具体的类型是未知的。List<?>所声明的就是所有类型都是可以的。但是List<?>并不等同于List<Object>。List<Object>实际上确定了List中包含的是Object及其子类，在使用的时候都可以通过Object来进行引用。而List<?>则其中所包含的元素类型是不确定。其中可能包含的是String，也可能是 Integer。如果它包含了String的话，往里面添加Integer类型的元素就是错误的。正因为类型未知，就不能通过new ArrayList()的方法来创建一个新的ArrayList对象。因为编译器无法知道具体的类型是什么。但是对于 List中的元素确总是可以用Object来引用的，因为虽然类型未知，但肯定是Object及其子类。考虑下面的代码：

public void wildcard(List<?> list) {

list.add(1);//编译错误

}

如上所示，试图对一个带通配符的泛型类进行操作的时候，总是会出现编译错误。其原因在于通配符所表示的类型是未知的。

因为对于List<?>中的元素只能用Object来引用，在有些情况下不是很方便。在这些情况下，可以使用上下界来限制未知类型的范围。 如 List<? extends Number>说明List中可能包含的元素类型是Number及其子类。而List<? super Number>则说明List中包含的是Number及其父类。当引入了上界之后，在使用类型的时候就可以使用上界类中定义的方法

3） 类型系统

在Java中，大家比较熟悉的是通过继承机制而产生的类型体系结构。比如String继承自Object。根据Lisk替换原则，子类是可以替换父类的。当需要Object类的引用的时候，如果传入一个String对象是没有任何问题的。但是反过来的话，即用父类的引用替换子类引用的时候，就需要进行强制类型转换。编译器并不能保证运行时刻这种转换一定是合法的。这种自动的子类替换父类的类型转换机制，对于数组也是适用的。 String[]可以替换Object[]。但是泛型的引入，对于这个类型系统产生了一定的影响。正如前面提到的List是不能替换掉List的。

引入泛型之后的类型系统增加了两个维度：一个是类型参数自身的继承体系结构，另外一个是泛型类或接口自身的继承体系结构。第一个指的是对于 List<String>和List<Object>这样的情况，类型参数String是继承自Object的。而第二种指的是 List接口继承自Collection接口。对于这个类型系统，有如下的一些规则：

相同类型参数的泛型类的关系取决于泛型类自身的继承体系结构。即List<String>是Collection<String> 的子类型，List<String>可以替换Collection<String>。这种情况也适用于带有上下界的类型声明。

当泛型类的类型声明中使用了通配符的时候，其子类型可以在两个维度上分别展开。如对Collection<? extends Number>来说，其子类型可以在Collection这个维度上展开，即List<? extends Number>和Set<? extends Number>等；也可以在Number这个层次上展开，即Collection<Double>和Collection<Integer>等。如此循环下去，ArrayList<Long>和 HashSet<Double>等也都算是Collection<? extends Number>的子类型。

如果泛型类中包含多个类型参数，则对于每个类型参数分别应用上面的规则。

四、

五、

一、内存泄漏

应该被回收的对象，却无法被回收。

产生原因：一个对象应该被回收，但是有另一个正在使用的对象存在该对象的引用，导致无法被GC回收。

造成的影响：

1. 应用卡顿，泄漏的内存影响了GC的内存分配，过多的内存泄漏会影响应用的执行效率。
2. OOM，大量的内存泄漏会导致OOM.

AndroidStudio自带的内存分析工具分析：Dump Java Heap:

二、内存溢出

当应用的heap资源超过了Dalvik虚拟机分配的内存就会内存溢出.

### GC工作原理

1. 垃圾回收机制

垃圾收集算法必须做两件事情：首先检测出垃圾对象，其次必须回收垃圾对象所使用的堆空间并还给程序。

从程序的主要运行对象(如静态对象/寄存器/栈上指向的堆内存对象等)开始检查引用链，当遍历一遍后得到上述这些无法回收的对象和他们所引用的对象链，组成无法回收的对象集合，而其他孤立对象（集）就作为垃圾回收。

区分活动对象和垃圾的两个基本方法是引用计数和跟踪。

引用计数收集器、跟踪收集器

引用计数收集器：

如果增加一个该对象的引用，那么该对象的引用计数加1，如果减少就减1，引用计数为0的对象应该被回收。

优点：执行的速度比较快，可以在程序运行中执行。

缺点：无法检测出循环引用的情况。导致不能被回收。

跟踪收集器（标记并清除）：

追踪从根节点开始的对象引用图。在追踪过程中对遇到的对象打标机，追踪结束后，未被标机的对象应该被收集。

针对推碎块，有两种方法：压缩、拷贝。

压缩：把活动的对象越过空闲区滑动到堆的一端，堆的另一端出现一个大的连续空闲区。所有被移动的对象的引用也被更新，指向新的位置。

拷贝：把所有活动对象移动到一个新的区域。一般被称为停止-拷贝，堆被分为两个区域，任何时候只能使用一个区域。区域用尽时，程序中止，堆被遍历，从一个区域拷贝到另一个区域。

优点：对象可以在根对象开始遍历的过程中随着发现而拷贝，不再有标记和清除的区分。

缺点：需要两倍大小的堆内存，但是任何时候只能使用其中一半。每一次收集时，所有的活动对象都会被拷贝。而有的对象生命周期很长，所以会被来回拷贝，消耗大量时间。

分代是Java垃圾收集的一大亮点，根据对象的生命周期长短，把堆分为3个代：Young，Old和Permanent，根据不同代的特点采用不同的收集算法。

年轻代：

采取拷贝收集的方法。GC较频繁。少量，多次。当年轻代的内存使用达到一定的阀值时，Minor Collection就被触发。

年老代：

年轻代的对象如果能够挺过数次收集，就会进入老人区。老人区使用标记整理算法。因为老人区的对象都没那么容易死的，采用复制算法就要反复的复制对象，很不合算，只好采用标记清理算法。

持久代：

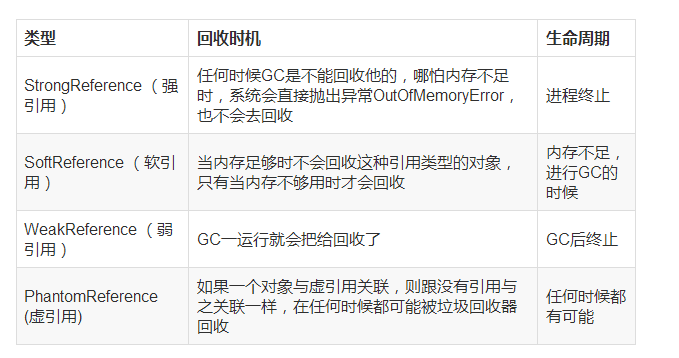
装载Class信息等基础数据。般情况下，持久代是不会进行GC的。

当年老代或者永久代的内存使用达到一定阀值时，一次基于所有代的GC就触发了，其特定是涉及范围广（量大），耗费的时间相对较长（较慢），但是频率比较低（次数少），称之为Major Collection(Full Collection)。通常，首先使用针对年轻代的GC算法进行年轻代的GC，然后使用针对年老代的GC算法对年老代和永久代进行GC。

自适应收集器：根据不同的情况选择不同的方法。

Android虚拟机的垃圾回收采用的是根搜索算法。GC会从根节点（GC Roots）开始对heap进行遍历。到最后，部分没有直接或者间接引用到GC Roots的就是需要回收的垃圾，会被GC回收掉。

1. GC过程与对象的引用类型关系



1. JVM内存分配
2. 静态的

静态存储区.

，内存在编译时就已经分配好，程序运行期间一直存在，存储静态数据、全局变量。

1. 栈式的

存储定义的变量，对象的引用。程序运行结束自动被释放。栈内存空间有限，大小由操作系统决定，Windows下栈为2M，连续的，不会产生碎片，运行效率高。

1. 堆式的

存储动态创建的对象。GC会根据内存的使用情况，对堆内存里的垃圾内存进行回收。堆内存是一块不连续的内存区域，如果频繁地new/remove会造成大量的内存碎片，GC频繁的回收，导致内存抖动，这也会消耗我们应用的性能。

1. ANR

Activity 5 s, BroadCast 10s。

1. 序列化

序列化：使得对象可以传输和存储。Java对象变成字节流的形式传出去，有的时候我们想要从一个字节流中恢复一个Java对象。

序列化方式：实现Serializable接口 和 Parcelable

Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

Parcelable是Android为我们提供的序列化的接口,Parcelable相对于Serializable的使用相对复杂一些,但Parcelable的效率相对Serializable也高很多。需要自己控制变量是否被序列化。

1. Service

启动Service 两种方法：通过Activity 启动Service startService();bindService();

startService:启动后一直运行，调用自身的stopSelf方法或者调用stopService()方法停止Service.service 生命周期：onCreate，onStartCommand,onDestory.

bindService: 生命周期：onCreate，onBind,onDestory.通过unbindService方法解绑。

多次启动Service不会创建新的实例。

Service和Activity通信：

通过IBinder,Activity 绑定Service时，会调用OnBind方法，该方法会返回一个IBInder对象，自定义继承一个IBinder对象，作为返回值。在Activity中实现ServiceConnection接口类，ServiceConnection接口中有两个方法：onServiceConnected 和 onServiceDisconnected，这两个方法分别会在Activity与Service建立关联和解除关联的时候调用。onServiceConnected方法中可以得到Service中返回的IBinder对象。然后就可以调用IBinder中的方法了。

创建的Service是运行在主线程中的，IntentService 是继承自 Service 并处理异步请求的一个类，在 IntentService 内有一个工作线程来处理耗时操作。当任务执行完后，IntentService 会自动停止，不需要我们去手动结束。因为自己创建Service需要手动结束。重写onHandleIntent方法。

1. Anroid框架



从下往上依次为

* 内核层：Linux 内核和各类硬件设备的驱动，这里需要注意的是，Binder IPC 驱动也是在这一层实现，比较特殊
* 硬件抽象层：封装「内核层」硬件驱动，提供可供「系统服务层」调用的统一硬件接口
* 系统服务层：提供核心服务，并且提供可供「应用程序框架层」调用的接口
* Binder IPC 层：作为「系统服务层」与「应用程序框架层」的 IPC 桥梁，互相传递接口调用的数据，实现跨进层的通讯
* 应用程序框架层：这一层可以理解为 Android SDK，提供四大组件，View 绘制体系等平时开发中用到的基础部件

1. JVM

JVM的主要任务是装载class文件，并且执行其中的字节码。JVM中包含一个类装载器，可以从程序和API中装载class文件。Java API中只有程序执行时需要的类才会被装载。字节码有执行引擎来执行。

Java中有两种方法：Java 方法和本地方法。Java方法是由Java语言编写，编译成字节码。本地方法是由其他语言编写的，编译成和处理器相关的及其代码。本地方法保存在动态链接库中。通过本地方法，java程序可以直接访问底层操作系统的资源。

一个Java程序可以使用两种类装载器：启动类装载器和用户定义的类装载器。

启动类装载器是JVM 实现的一部分。也被称为原始类装载器、系统类装载器、默认类装载器、

用户定义的类装载器能够使用用户自定义的方式来装载类，比如从网络上下载class文件。用户定义的类装载器不是JVM实现的部分，它能够用Java编写，被编译为class文件，能够被虚拟机装载，实际上只是运行中的Java应用程序可执行代码的一部分。

被装载的类默认情况下只能看到被同一个类装载器装载的别的类。

Java class文件是可以运行在任何支持Java虚拟机的硬件平台和操作系统上的二进制文件。

Java API 是运行库的集合，它提供一套访问主机系统资源的标准方法。，Java API调用本地方法。

Java虚拟机包含三种含义：抽象规范、一稿具体的实现、一稿运行中的虚拟机实例。

一稿运行时的Java虚拟机的任务是运行一个Java程序。如果在同一台计算机上同时运行三个Java程序，将得到三个Java虚拟机实例，每个Java程序都运行与自己的Java虚拟机实例中。

Java虚拟机实例通过调用某个初始类的main（）方法来运行一稿Java程序。

Java程序初始类中的main方法，将作为该程序初始线程的起点，其他任何线程都是由这个初始线程启动的。

在Java虚拟机中有两种线程：守护线程和非守护线程。守护线程通常是由虚拟机自己使用的，比如执行垃圾收集任务的线程。但是，Java程序也可以把它创建的任何线程标记为守护线程。Java初始线程（main）是非守护线程。只要还有任何非守护线程在运行，那么这个Java程序也在运行。当所有的非守护线程都终止时，虚拟机实例将自动退出。

某些运行时数据区是由查询中所有线程共享的，有一些只能由一个线程拥有。每个Java虚拟机实例都有一个方法区和一个堆，它们是由该虚拟机实例中的所有线程共享的，当虚拟机装载一个class文件时，它会从这个class文件包含的二进制数据中解析类型信息。然后把这些类型信息放到方法区中。当程序运行时，虚拟机会把所有该进程在运行时创建的对象都放在堆中。

当每一个新线程被创建时，她都将得到自己的PC寄存器（程序计数器）以及一个Java栈。如果线程正在执行的是一个Java方法，那么PC寄存器的值将总是指示下一条将被执行的指令。

**运行时数据区**：

**方法区**：虚拟机实例所有线程共享，存储类型信息（全限定名，类类型or接口类型，类型的修饰符（public，abstract，final），字段名，字段类型，方法名，方法返回类型，参数类型、静态变量等等），。

**堆**：虚拟机实例所有线程共享，存储动态创建的对象。

**Java栈**：线程独享，存储该线程中Java方法调用的状态，包括局部变量，被调用时传进来的参数，返回值，以及运算的中间结果。

**PC寄存器**：线程独享，指示下一条将被执行的指令。

**本地方法栈**：存储本地方法的调用状态。

Java数据类型：基本类型、引用类型。

虽然Java虚拟机也把boolean看做基本类型，但是编译为字节码时，会用int或byte来表示boolean。在Java虚拟机中，false是由整数0来表示的，所有非0整数都表示true。涉及Boolean值的操作则会使用int。Boolean数组是当byte数组来访问。

Java虚拟机有一个只在内部使用的基本类型：returnAddress，这个基本类型被用来实现Java程序中的finally字句。

用户自定义的类装载器必须派生自ClassLoader类。

对于每一个被装载的类型，Java虚拟机都会为它创建一个Class类的实例来代表该类型。

1. Activity生命周期

onCreate,onStart,onResume,onPause,onStop,onDestroy.

onStart: Activity已经显示出来，但是我们看不到，相当于是在后台。

OnResume: Activity 已经可见了，OnResume 的时候Activity才显示到前台。

onPause: onPause 必须先执行完，新Activity的onResume才能执行。

如果新Activity采用了透明主题，那么当前Activity不会回调onStop。

OnCreate 对应 onDestroy; onStart 对应onStop, onPause 对应onResume.

默认情况下，手机横竖屏切换会导致Activity被消耗并且重新创建。

通过配置configChanges=orientation属性可以防止系统重新创建Activity。

Activity在异常情况下终止，系统会调用onSaveInstanceState来保存当前Activity的状态。调用时机在onStop之前。与onPause没有时序关系。当Activity被重新创建后，系统会调用onRestoreInstanceState，并把消耗时onSaveInstanceState方法所保存的Bundle对象作为参数传递给onRestoreInstanceState和onCreate方法。onRestoreInstanceState调用时机在onStart之后。

每一个View都有onSaveInstanceState和onRetorInstanceState方法。

Activity的启动模式：

Standard：标准模式。

SingleTop：栈顶复用，如果Activity位于栈顶，则不会被重新创建，否则会重新创建。同时会调用onNewIntent方法。

SingTask：栈内复用，如果所需的任务栈存在，如果Activity栈内存在，则将Activity调到栈顶。如果不存在则创建。如果所需的任务栈不存在，则创建任务栈，再创建Activity 压入栈。同时会执行onNewIntent方法。

SingleInstance：单实例模式，加强的singleTask。为启动的Activity创建一个新的任务栈，后续不会创建新的Activity。

IntentFilter中的过滤信息有：action，category，data。

1. IPC 机制

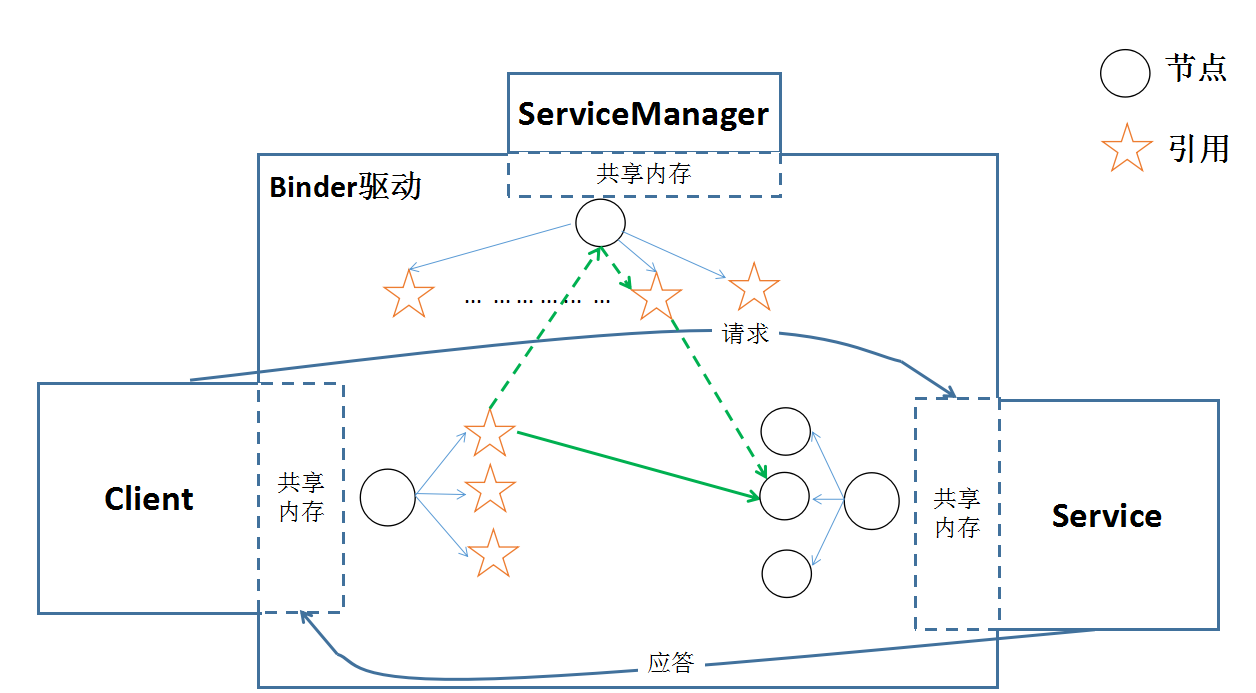
开启多进程：指定android:process属性。

Android为每一个应用分配了一个独立的虚拟机，为每一个进程都分配一个独立的虚拟机，不同的虚拟机有不同的地址空间，这就导致在不同的虚拟机中访问同一个对象会产生多个副本。不同的进程的组件会有独立的虚拟机，Application，以及内存空间。

实现跨进程的方式：Intent传递数据，共享文件和sharepreference，基于Binder的Messenger和AID，Socket。

Binder机制：

客户端持有远程进程的某个对象引用，然后调用引用类中的函数，远程进程的函数就执行了。



服务端跨进程的类都要继承Binder类。我们所持有的Binder引用(即服务端的类引用)并不是实际真实的远程Binder对象，我们的引用在Binder驱动里还要做一次映射。也就是说，设备驱动根据我们的引用对象找到对应的远程进程。客户端要调用远程对象函数时，只需把数据写入到Parcel，在调用所持有的Binder引用的transact()函数，transact函数执行过程中会把参数、标识符（标记远程对象及其函数）等数据放入到Client的共享内存，Binder驱动从Client的共享内存中读取数据，根据这些数据找到对应的远程进程的共享内存，把数据拷贝到远程进程的共享内存中，并通知远程进程执行onTransact()函数，这个函数也是属于Binder类。远程进程Binder对象执行完成后，将得到的写入自己的共享内存中，Binder驱动再将远程进程的共享内存数据拷贝到客户端的共享内存，并唤醒客户端线程。